Amélioration des performances de supports d'exécution à tâches à l'aide de broadcasts dynamiques

Philippe Swartvagher

Compas'2020 : Parallélisme / Architecture/ Système/ Temps Réel
MILC - Lyon, France, du 30 juin au 3 juillet 2020

Amélioration des performances de supports d'exécution à tâches à l'aide de broadcasts dynamiques

Philippe SWARTVAGHER

Inria Bordeaux – Sud-Ouest,
200 Avenue de la Vieille Tour
33405 Talence - France
philippe.swartvagher@inria.fr

Résumé
Les supports d'exécution à tâches sont apparus dans le monde du calcul haute performance pour gérer plus facilement les architectures hétérogènes, mieux passer à l'échelle et atteindre ainsi de meilleures performances. Le passage à l'échelle sur de nombreux nœuds de calcul fait apparaître les communications réseau comme un goulot d'étranglement pour les performances. Les supports d'exécution à tâches doivent parfois envoyer une même donnée vers plusieurs nœuds. L'optimisation de ce genre de communications passe habituellement par un appel à une fonction dédiée, comme `MPI_Bcast`. Cependant, `MPI_Bcast` est difficilement utilisable dans les supports d'exécution à tâches : les nœuds doivent savoir qu'ils ne sont pas les seuls à recevoir la donnée en question et ils doivent connaître tous les autres nœuds destinataires de cette donnée, pour pouvoir faire appel à `MPI_Bcast`.
Pour pallier ces contraintes tout en conservant un algorithme de diffusion optimisé, nous proposons une solution, appelée *broadcasts dynamiques*, où les nœuds découvrent, de façon transparente pour l'utilisateur, qu'ils sont impliqués dans un broadcast seulement lorsqu'ils reçoivent les données et n'ont pas besoin de connaître tous les nœuds du broadcast.
Cette solution montre des améliorations des performances allant jusqu'à 30 % sur des décompositions de CHOLESKY.

Mots-clés : communications, collective, broadcast, support d'exécution à tâches

---

1. Introduction

Le passage à l'échelle des applications parallèles distribuées est limité, entre autre, par les synchronisations entre les nœuds. Les supports d'exécution à tâches ordonnancent donc l'exécution des tâches sur les différents nœuds sans synchronisation entre les nœuds. Cette absence de synchronisation, qui réduit le temps d'exécution des applications, doit cependant aussi se retrouver dans la gestion des communications entre nœuds.
Les supports d'exécution peuvent être amenés à transmettre des données présentes sur un nœud vers plusieurs autres nœuds. Pour ce genre d'opérations, les bibliothèques reposant sur le standard MPI [11] proposent les fonctions `MPI_Bcast` ou `MPI_Ibcast`, qui utilisent alors un routage des données efficace. Cependant, les contraintes imposées par `MPI_Bcast` rendent son utilisation difficile et peu efficace avec les supports d'exécution à tâches.

À travers cet article, nous proposons un algorithme appelé *broadcast dynamique* qui permet d'employer des algorithmes de routage optimisés, tout en s'adaptant aux besoins des supports d'exécution à tâches. Nous avons implémenté notre algorithme dans les bibliothèques StarPU [4] et NewMadeleine [5] et montrons que les broadcasts dynamiques améliorent les performances de décompositions de Cholesky jusqu'à 30 %.
L'article suit le plan suivant : la Section 2 présente en détail l'incompatibilité entre les supports d'exécution à tâches et `MPI_Bcast`, la section 3 décrit la solution que nous proposons, la section 4 montre les performances obtenues, la Section 5 explore les travaux similaires et finalement la Section 6 conclut l'article.

2. Broadcasts et supports d'exécution à tâches

L'utilisateur d'un support d'exécution à tâches décompose son application en tâches et indique ensuite les dépendances entre ces tâches. Le support d'exécution génère ainsi un graphe acyclique orienté (un DAG : *Directed Acyclic Graph*) où les sommets sont les tâches et les arêtes les dépendances entre ces tâches. C'est ce graphe qui lui permet d'ordonnancer les tâches sur les différentes unités de calcul dont il dispose (CPU, GPU ou autre accélérateur). Sur des systèmes distribués, le support d'exécution répartit les tâches entre les différents nœuds. Lorsqu'une arête du graphe lie deux tâches présentes sur deux nœuds différents, le support d'exécution se charge de transférer d'un nœud à l'autre, par le réseau, les données nécessaires pour l'exécution de la tâche cible de l'arête.
Une donnée présente sur un nœud peut être nécessaire à plusieurs tâches réparties sur des nœuds différents (du point de vue du graphe : un sommet a plusieurs arêtes sortantes menant vers des sommets situés sur des nœuds différents). Ce type de diffusion de données est un *multicast*, ou un *broadcast* dans le jargon MPI. C'est un type de communications *collectives*, à l'inverse des communications *point-à-point* qui n'incluent qu'un émetteur et un récepteur.
La manière naïve d'effectuer un broadcast est de charger le nœud source d'envoyer les données à tous les destinataires, par une succession de communications point-à-point. La durée d'un broadcast est de cette façon linéaire en nombre de destinataires. Généralement, les bibliothèques de communication utilisent de meilleurs algorithmes pour les broadcasts [12, 15, 13], comme les arbres binaires, les arbres binomiaux ou des arbres pipelinés, qui offrent une complexité logarithmique en nombre de destinataires. C'est pour profiter de ces bons algorithmes que l'usage de fonctions comme `MPI_Bcast` est recommandé pour réaliser des broadcasts.
Cependant, pour les supports d'exécution à tâches distribués, qui construisent leurs graphes de tâches dynamiquement et qui n'ont pas une vision globale sur l'ordonnancement de chaque nœud et la distribution des données, `MPI_Bcast` est difficile à utiliser, et même inefficace, pour plusieurs raisons :

- la détection des broadcasts n'est pas évidente. Les seules informations dont dispose le support d'exécution pour inférer les communications entre nœuds proviennent du graphe de tâches. Lorsque ce graphe est construit dynamiquement, il n'est pas possible de savoir si toutes les tâches ont été soumises et donc si la liste des destinataires est complète ou s'il faut encore attendre d'éventuels destinataires avant de démarrer le broadcast.
- tous les nœuds participants à un broadcast, émetteur comme récepteurs, doivent faire appel à la fonction `MPI_Bcast`. Il est donc nécessaire que le support d'exécution sache si les données vont lui parvenir par une communication point-à-point ou par un broadcast. Cependant, chaque nœud n'ayant qu'une vision locale du graphe de tâches, il n'est pas possible pour le support d'exécution de connaître cette information.

— l'appel à `MPI_Bcast` nécessite un *communicateur*, une structure de données contenant tous les nœuds impliqués dans le broadcast. Cela signifie qu'avant même de lancer le broadcast, tous les nœuds impliqués dans le broadcast doivent tous se connaître. À cause de la vision locale du graphe de tâches propre à chaque nœud, la seule façon pour les nœuds de connaître les autres nœuds impliqués dans un broadcast serait que le nœud source leur transmette cette information : c'est un autre broadcast !

Finalement, le coût en terme de performances pour satisfaire les contraintes de `MPI_Bcast` risque de s'avérer supérieur au gain apporté par l'utilisation de `MPI_Bcast`. La problématique générale consiste à pouvoir exploiter des algorithmes de broadcast efficaces, sans que tous les nœuds des broadcasts n'aient à se connaître auparavant et qu'ils n'aient pas besoin de savoir à l'avance qu'ils vont être impliqués dans un broadcast. La suite de cet article explique comment nous sommes parvenus à cet objectif.

3. Notre solution : les broadcasts dynamiques

Cette partie présente les différents éléments mis en place pour détecter efficacement les broadcasts et faire des broadcasts sans appel explicite de la part des récepteurs et sans que tous les nœuds destinataires n'aient à se connaître.

3.1. StarPU et NewMadeleine

Bien que notre solution puisse s'adapter à n'importe quel support d'exécution, nous avons utilisé StarPU avec NewMadeleine pour gérer les communications.

StarPU [4] est un support d'exécution à tâches pour architectures hétérogènes. Il repose sur le *Sequential Task Flow* qui permet à l'utilisateur de soumettre des tâches à StarPU, celui-ci se chargeant de construire le graphe de tâches et d'ordonnancer leurs exécutions sur les unités de calcul disponibles. Dans sa version distribuée [3], les tâches sont réparties sur les différents nœuds disponibles, les communications nécessaires sont détectées à partir du graphe de tâches et sont réalisées par une bibliothèque tierce : une bibliothèque MPI ou NewMadeleine.

NewMadeleine [5] est une bibliothèque de communication pour le HPC. Plutôt que d'avoir une activité imposée par l'application, NewMadeleine se repose sur l'activité des interfaces réseau et applique ainsi des stratégies sur les données à envoyer ou à recevoir : notamment l'agrégation ou la prise en compte des priorités. Son paradigme événementiel permet une grande réactivité.

3.2. Détection des collectives

Lorsque le support d'exécution détecte qu'une arête du graphe de tâches passe d'un nœud à un autre, il doit transférer la donnée représentée par cette arête au nœud qui va en avoir besoin. Le support d'exécution soumet alors une opération d'envoi. Cette opération d'envoi peut être déclarée avant que les données à envoyer ne soient prêtes. Les données sont alors envoyées dès qu'elles deviennent disponibles.

Dans le cas des broadcasts, il y a plusieurs arêtes partant d'un même sommet dans le graphe de tâches. Il y a alors pour chaque arête une opération d'envoi. Notre détection de broadcasts agrège les opérations d'envoi pour une même donnée tant qu'elle n'est pas disponible : lorsqu'une nouvelle opération d'envoi pour une donnée encore indisponible est soumise, le destinataire de l'envoi est ajouté à une liste de destinataires (s'il n'y est pas déjà). Lorsque la donnée devient disponible, si la liste de destinataires contient plus d'un destinataire, un broadcast est déclenché, sinon la donnée est transmise par une simple requête point-à-point.

Cette façon de faire repose sur le fait que, la plupart du temps, toutes les tâches sont soumises

rapidement au début de l'exécution de l'application. L'exécution des tâches étant plus lente que leur soumission au support d'exécution, la majorité des opérations d'envois sont postées avant que les données ne soient disponibles. Malgré cela, cette méthode n'empêche pas d'avoir seulement une partie des destinataires d'un broadcast (des envois se sont ajoutés après que la donnée soit devenue disponible). Ces cas de figure ne sont pas très importants, puisque cela signifie que l'ordonnanceur n'avait pas besoin d'envoyer cette donnée plus tôt.

3.3. Algorithme de broadcast dynamique

Notre implémentation exécute un algorithme de broadcast optimisé, tout en utilisant des messages actifs qui contiennent, en plus des données à transmettre, des informations sur le routage obtenues par l'algorithme de broadcast.

Les algorithmes de broadcasts optimisés sont nombreux [16, 12, 15, 13] et bien connus. Bien que notre implémentation puisse utiliser n'importe quel algorithme de broadcast, nous avons choisi d'utiliser la diffusion par arbre binomial, car c'est l'algorithme qui offre le meilleur compromis pour améliorer les performances, quels que soient le nombre de destinataires et la taille des données à transmettre. L'arbre binomial a une complexité logarithmique en nombre de destinataires. Avec l'arbre binomial, chaque nœud qui reçoit une donnée se met également à l'envoyer à des nœuds qui ne l'ont pas encore reçue ; et ce, jusqu'à ce que tous les destinataires aient reçu la donnée.

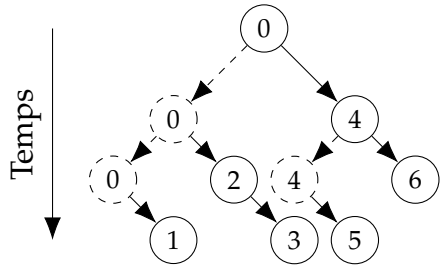

FIGURE 1 – Exemple d'arbre binomial avec 5 destinataires. Les étiquettes des sommets correspondent au rang des différents noeuds dans le communicateur global. Chaque niveau de l'arbre est une étape de l'algorithme.

La Figure 1 illustre un broadcast du nœud 0 vers 5 destinataires : le nœud 0 commence par envoyer la données au nœud 4 ; puis pendant que 0 envoie à 2, 4 envoie à 6 ; finalement, 0 envoie à 1, 2 envoie à 3 et 4 envoie à 5.

Puisque les nœuds récepteurs ne peuvent pas savoir qu'ils vont faire partie d'un broadcast et n'ont donc aucune information concernant le broadcast, toutes les informations nécessaires sont stockés dans l'en-tête du message transportant les données.

Les informations stockées dans l'en-tête sont notamment les nœuds auxquels le nœud recevant le message devra faire suivre les données : dans l'arbre binomial, il s'agit de la liste des nœuds situés sous le nœud qui va recevoir le message. Ainsi, lorsqu'un nœud reçoit ce type de message, il reconstruit un arbre binomial avec la liste des nœuds et envoie des messages de la même forme aux nœuds indiqués par l'arbre binomial.

Ces messages de broadcasts étant imprévisibles pour les récepteurs et faisant l'objet d'un traitement spécial (l'envoi des données à d'autres nœuds avant de rendre les données accessibles à l'application), ils sont directement traités par la bibliothèque de communication, en passant par un canal dédié. Lorsqu'un message arrive par ce canal, les données sont transférées aux autres nœuds, puis la réception normale (point-à-point) attendue par l'application est notifiée

que les données sont bien arrivées. Ainsi la réception de données par des broadcasts est transparente pour l'application. Ces messages de broadcasts étant des messages actifs, ils utilisent l'interface `rpc` (*Remote Procedure Call*) de NEWMADELEINE, qui permet d'exécuter des instructions (faire suivre les données à d'autres nœuds, par exemple) dès qu'un message est reçu, sans appel spécifique de la part de l'application.

4. Évaluation des performances

Nous avons évalué l'impact des broadcasts dynamiques en mesurant les performances de décompositions de CHOLESKY. Cette opération matricielle consiste, pour toute matrice symétrique définie positive A, à calculer la matrice triangulaire inférieure L, telle que $A = LL^T$. Cet algorithme est un bon candidat pour mesurer l'impact des broadcasts, puisqu'il génère beaucoup de broadcasts. Nous utilisons la bibliothèque CHAMELEON [2], qui utilise déjà STARPU, pour exécuter cette opération matricielle.
Nos expériences ont été menées sur la machine `inti` du CEA. Ce cluster dispose de nœuds avec deux processeurs Xeon E5-2680 cadencés à 2.7 GHz avec chacun 16 coeurs. 64 Go de RAM équipent chaque nœud, ainsi que des cartes réseau Connect-IB *InfiniBand* QDR (MT27600). La version de OpenMPI par défaut sur ce cluster étant OpenMPI 2.0, nous avons compilé nous-mêmes la version 4.

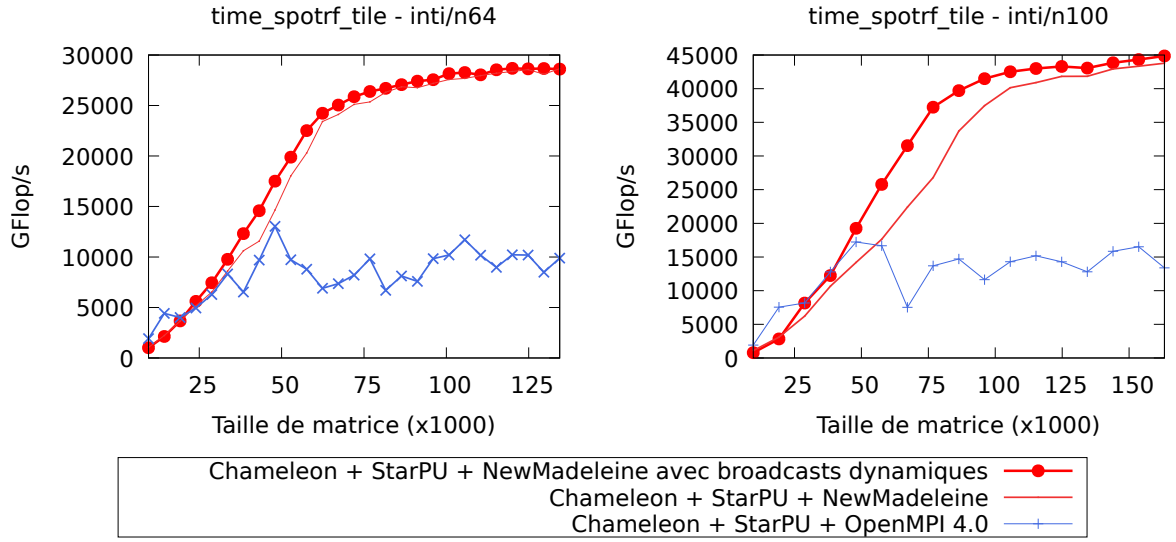

FIGURE 2 – Performances de la factorisation de CHOLESKY sur le cluster `inti` : sur 64 nœuds (1024 coeurs) à gauche et sur 100 nœuds (1600 coeurs) à droite. Un seul processus MPI est exécuté par nœud. Chaque point correspond à la moyenne de deux exécutions.

La Figure 2 montre les résultats obtenus. Nous avons comparé les performances quand STARPU utilise NEWMADELEINE avec les broadcasts dynamiques, NEWMADELEINE sans broadcast dynamique et finalement OpenMPI 4 (sans broadcast, donc), qui nous sert de référence. L'écart entre NEWMADELEINE et OpenMPI s'explique par d'autres raisons [8], dépassant le cadre de cet article.
Dans les deux cas, les broadcasts dynamiques améliorent les performances générales, notamment pour les petites matrices. Sur 64 nœuds, les broadcasts dynamiques améliorent jusqu'à 20 % les performances et jusqu'à 30 % sur 100 nœuds. Le nombre et les tailles des broadcasts augmentant avec le nombre de nœuds, il est logique que l'impact soit plus important sur un

plus grand nombre de nœuds. Pour les plus grandes matrices, les communications ont moins d'impact puisque chaque nœud dispose toujours de suffisamment de tâches à exécuter avant de devoir attendre que des données arrivent par le réseau.

5. Travaux connexes

Les algorithmes de broadcasts ont déjà fait l'objet de nombreux travaux [16, 12, 15, 13] et ne sont pas le sujet principal de cet article, notre implémentation pouvant s'appliquer à n'importe quel algorithme de broadcast.
PARSEC [7] est un support d'exécution, dont le graphe de tâches est décrit à l'aide d'une représentation algébrique (le *Parameterized Task Graph*). Cette représentation offre l'avantage d'avoir un coût en mémoire très faible, même avec de nombreuses tâches. Ainsi, chaque nœud connaît le graphe de tâches entier et peut ainsi facilement répondre aux contraintes imposées par `MPI_Bcast`, décrite dans la Section 2. En pratique, PARSEC implémente un algorithme d'arbre binomial et un algorithme d'arbre chaîné à l'aide de communications point-à-point MPI. CLUSTERSS [14] est un support d'exécution dont la version distribuée repose sur un modèle maître-esclave : un nœud a une vision globale du graphe de tâches et distribue le travail aux autres nœuds. Cette approche centralisée permet de facilement détecter les broadcasts, bien qu'aucune publication ne fasse mention de l'optimisation des broadcasts. LEGION [6] est un support d'exécution axé sur la localité des données. Sa politique d'ordonnancement par défaut est le vol de tâche, même entre différents nœuds. Il n'y pas de détail concernant l'optimisation des communications, mais généralement le vol de travail est fait sans synchronisation entre les nœuds qui nécessiteraient la même donnée. CHARM++ [1] et HPX [10] sont tous les deux des systèmes pour les applications parallèles et distribuées. Ils proposent des mécanismes de broadcasts, mais que l'utilisateur doit appeler explicitement. KAAPI [9] est une bibliothèque d'ordonnancement de tâches, basée sur le vol de travail. La communication entre nœuds se fait à l'aide de messages actifs. Il n'est pas fait mention d'optimisations particulières des broadcasts.
Finalement, aucun des autres supports d'exécution à tâches n'avait exactement les mêmes contraintes que celles présentées par STARPU : chaque nœud n'a qu'une vision locale du graphe de tâche et il n'y pas de synchronisation entre les nœuds à propos de l'ordonnancement des tâches, ni de nœud maître qui pourrait avoir une vue d'ensemble et facilement traiter les broadcasts.

6. Conclusion et perspectives

Lorsqu'il est nécessaire de transmettre une donnée à plusieurs nœuds, l'utilisation de `MPI_Bcast` n'est guère possible pour les supports d'exécution à tâches où chaque nœud n'a qu'une vision locale du graphe de tâches. Pour pouvoir profiter d'algorithmes de broadcast efficaces, nous avons développé des *broadcasts dynamiques* qui permettent de stocker les informations relatives au broadcast dans l'en-tête des données à transmettre et qui sont traités en marge de l'application par la bibliothèque de communication, pour être complètement transparents pour l'application. Ces broadcasts dynamiques permettent d'améliorer les performances générales de décompositions de CHOLESKY jusqu'à 30 %, démontrant au passage l'importance de l'optimisation des broadcasts dans les supports d'exécution à tâches distribués.
Il reste maintenant à implémenter d'autres algorithmes de broadcast, pour utiliser le plus efficace selon la situation. Il sera également intéressant d'implémenter les broadcasts dynamiques pour les bibliothèques MPI, par dessus les opérations points-à-points. Cependant, le défi le plus important reste l'analyse précise des performances obtenues lors de l'utilisation de sup-

ports d'exécution à tâches sur de nombreux nœuds.

Remerciements
This work is supported by the Agence Nationale de la Recherche, under grant ANR-19-CE46-0009. This work is supported by the Région Nouvelle-Aquitaine, under grant 2018-1R50119 *HPC scalable ecosystem*. Experiments presented in this paper were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see `https://www.plafrim.fr/`).
This work was granted access to the HPC resources of CINES under the allocation 2019- A0060601567 attributed by GENCI (Grand Equipement National de Calcul Intensif).
L'auteur tient à remercier Alexandre DENIS, Emmanuel JEANNOT, Samuel THIBAULT, Olivier AUMAGE et Nathalie FURMENTO pour leurs conseils et leur aide précieuse apportés lors de ces travaux.

Bibliographie

1. Acun (B.), Gupta (A.), Jain (N.), Langer (A.), Menon (H.), Mikida (E.), Ni (X.), Robson (M.), Sun (Y.), Totoni (E.) et al. – Parallel programming with migratable objects : Charm++ in practice. – In *SC'14 : Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 647–658. IEEE, 2014.
2. Agullo (E.), Augonnet (C.), Dongarra (J.), Ltaief (H.), Namyst (R.), Thibault (S.) et Tomov (S.). – Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. *In : GPU Computing Gems*, éd. par mei W. Hwu (W.). – Morgan Kaufmann, septembre 2010.
3. Agullo (E.), Aumage (O.), Faverge (M.), Furmento (N.), Pruvost (F.), Sergent (M.) et Thibault (S.). – Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *IEEE Transactions on Parallel and Distributed Systems*, 2017.
4. Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *CCPE - Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009*, vol. 23, février 2011, pp. 187–198.
5. Aumage (O.), Brunet (E.), Furmento (N.) et Namyst (R.). – NewMadeleine : a Fast Communication Scheduling Engine for High Performance Networks. – In *Workshop on Communication Architecture for Clusters (CAC 2007)*, Long Beach, California, United States, mars 2007.
6. Bauer (M.), Treichler (S.), Slaughter (E.) et Aiken (A.). – Legion : Expressing locality and independence with logical regions. – In *SC '12 : Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, Nov 2012.
7. Bosilca (G.), Bouteiller (A.), Danalis (A.), Herault (T.), Luszczek (P.) et Dongarra (J.). – Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach. *Scalable Computing and Communications : Theory and Practice*, 2013-03 2013, pp. 699–735.
8. Denis (A.). – Scalability of the NewMadeleine Communication Library for Large Numbers of MPI Point-to-Point Requests. – In *CCGrid 2019 - 19th Annual IEEE/ACM International Symposium in Cluster, Cloud, and Grid Computing*, Larnaca, Cyprus, mai 2019.
9. Gautier (T.), Besseron (X.) et Pigeon (L.). – KAAPI : A thread scheduling runtime system for data flow computations on cluster of multi-processors. – In *2007 international workshop on Parallel symbolic computation*, pp. 15–23, Waterloo, Canada, juillet 2007. ACM.
10. Kaiser (H.), Brodowicz (M.) et Sterling (T.). – Parallex an advanced parallel execution model for scaling-impaired applications. – In *2009 International Conference on Parallel Processing Workshops*, pp. 394–401, Sep. 2009.

11. MPI Forum. – MPI : A Message-Passing Interface Standard Version 3.1. 2015.
12. Pješivac-Grbović (J.), Angskun (T.), Bosilca (G.), Fagg (G. E.), Gabriel (E.) et Dongarra (J. J.). – Performance analysis of MPI collective operations. *Cluster Computing*, vol. 10, n2, 2007, pp. 127–143.
13. Sanders (P.), Speck (J.) et Träff (J. L.). – Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Computing*, vol. 35, n12, 2009, pp. 581 – 594. – Selected papers from the 14th European PVM/MPI Users Group Meeting.
14. Tejedor (E.), Farreras (M.), Grove (D.), Badia (R. M.), Almasi (G.) et Labarta (J.). – A high-productivity task-based programming model for clusters. *Concurrency and Computation : Practice and Experience*, vol. 24, n18, 2012, pp. 2421–2448.
15. Träff (J. L.) et Ripke (A.). – Optimal broadcast for fully connected processor-node networks. *Journal of Parallel and Distributed Computing*, vol. 68, n7, 2008, pp. 887 – 901.
16. Wickramasinghe (U.) et Lumsdaine (A.). – A survey of methods for collective communication optimization and tuning. *CoRR*, vol. abs/1611.06334, 2016.